Problem Seti 7 Çözümleri

Problem 7-1. Edit distance (Biçimlendirme mesafesi)

Bu problemde biçimlendirme mesafesini hesaplamak için bir program yazacaksınız. Bu çözülmesi zorunlu bir problem. Bunu yapıp teslim etmezseniz yarı-yıl notunuzu önemli ölçüde olumsuz olarak etkileyecektir. Bu programlama ödevini bir an önce başlamanızı öneririz. Çünkü tüm detayları programın içine doğru yerleştirmek, umduğunuzdan fazla zaman alabilir.

Birçok kelime işlemcisi ve anahtar sözcük arama motorunun bir yazım düzeltme özelliği vardır. Eğer bir *x* sözcüğünü yanlış yazarsanız, kelime işlemcisi veya arama motoru bir *y* düzeltmesi önerebilir. *y* düzeltmesi, *x*' e yakın bir sözcük olmalıdır. Yazımdaki 2 harf dizgisi arasındaki benzerliği ölçmenin bir yolu, ''edit distance'' yani biçimlendirme mesafesidir. Biçimlendirme mesafesi kavramı, başka alanlarda da yararlıdır. Örneğin; biyologlar,DNA veya protein dizileri arasındaki benzerliği, biçimlendirme mesafesi kullanarak belirtirler.

x [1.. m] ve *y [1.. n]* gibi 2 harf dizgisinin biçimlendirme mesafesi *x[1.. m]* dizgisinin *y[1..* n][1] dizgisine dönüştüren''dönüştürme işlemleri'' dizisinin en az maliyetli olanı olarak tanımlanır (aşağıda tanımlanmıştır).Dönüşüm işlemlerinin etkisini tanmlamak için ara sonuçları saklayan bir *z[1.. s]*dizgisi kullanırız. Dönüşüm dizisinin başında *s = m ve z[1.. s] = x[1.. m]* (yani biz *x[1.. m]* dizgisiyle başlarız). Dönüşüm dizisinin sonunda elimizde *s = n ve z[1.. s] = y[1.. n]* olmalıdır. (yani hedefimiz, *y[.. n]* dizgisine dönüşmektir*).* Dönüşüm boyunca z dizgisinin uzunluğu olan *s*' yi ve imleç konumu olan *i*' yi (yani *z* dizgisinin bir anahtar listesini) koruruz. Dönüşüm boyunca $1 \leq i \leq s+1$ değişmezi her zaman geçerlidir. (imleçin, *z* dizgisinin sonundan bir adım öteye gidebildiğine ve böylece dizginin sonuna ekleme yapabildiğine dikkat edin).

Her dönüşüm işlemi, *z* dizgisini, *s* boyutunu ve *I* imleç konumunu değiştirebilir. Her dönüşüm işleminin ilgili bir maliyeti vardır. Dönüşüm işlemleri dizisinin maliyeti, dizideki bağımsız işlemlerin maliyetlerinin toplamına eşittir. Biçimlendirme probleminin hedefi *x[1.. m]* 'yi *y[1.. n]*'ye dönüştürecek dönüşüm işlemleri dizisini, en az maliyetli olanını bulmaktır.

[1] Burada bir metin dizgisini, harflerin bir dizilimi olarak görüyoruz. Bağımsız harfler sabit zamanda işlenebilir

5 dönüşüm işlemi vardır:

İşlem	*Maliyet*	*Etki*
Left(sol)	0	*i*=1 ise birşey yapma, değilse i ← *i*-1
Right(sağ)	0	*i*=s+1 ise birşey yapma, değilse i ← *i*+1
Replace(değiştir)	4	*i* = s+1 ise birşey yapma, değilse imleçin altındaki harfi c karakteriyle değiştir ve z[*i*] ← c yaptıktan sonra *i'* yi arttır.
Delete(sil)	2	*i* = s+1 ise birşey yapma, değilse imleçin altındaki *c* harfini z [*i*..s] ← z[*i*+1..s+1] yaptıktan sonra *s*' yi azalt. İmleç konumu *i* değişmeyecek.
Insert(araya yerleştir)	3	*c* harfini, *s'* yi arttırarak, *z* dizgisinde araya yerleştir ve z[i+1..s] ← z[i..s-1] ile z[i] ← c yap; sonra da *i*' yi arttır.

Örnek olarak kaynak dizgisi "***algorithm***" i, hedef dizgisi "***analysis***" e dönüştürme yollarından biri Tablo 1' deki işlemler dizisidir; burada altı çizili harf ***i*** imleçinin konumunu gösterir. Tablo 1' deki çözüm tek çözüm değildir, ***algorithm***'i ***analysis***'e dönüştüren birçok dönüşüm işlemi dizisi vardır ve bunların bazıları daha fazla bazıları da daha az maliyet çıkarır.

İşlem T	*z*	Maliyet	*Toplam*
ilk dizgi	algorithm	0	0
sağ	algorithm	0	0
sağ	algorithm	0	0
y ile değiştir	alyorithm	4	4
s ile değiştir	alysrithm	4	8
i ile değiştir	alysiithm	4	12
s ile değiştir	alysisthm	4	16
sil	alysishm	2	18
sil	alysism	2	20
sil	alysis	2	22
sol	alysis	0	22
sol	alysis	0	22
sol	alysis	0	22
sol	alysis	0	22
sol	alysis	0	22
n'yi Ara.Yer.	anlysis	3	25
a'yı Ara.Yer.	analysis	3	28

Tablo 1: Algorithm' i analysis'e dönüştürmek.

(a) Algorithm'i analysis' e 'sol' işlemini kullanmadan da dönüştürmek mümkündür. Tablo 1 ile maliyeti aynı olan ancak 'sol' işlemini kullanmayan bir işlemler dizisi verin.

Çözüm:

İşlem		Maliyet	*Toplam*
ilk dizgi	**algorithm**	**0**	0
sağ	algorithm	**0**	0
n'yi ara.yer.	anlgorithm	**3**	3
a'yı ara.yer.	analgorithm	**3**	6
sağ	analgorithm	**0**	6
y ile değiştir	analyorithm	**4**	10
s ile değiştir	analysrithm	**4**	14
i ile değiştir	analysiithm	**4**	18
s ile değiştir	analysisthm	**4**	22
sil	analysishm	**2**	24
sil	analysism	**2**	26
sil	**analysis**	**2**	28

Tablo 2: Sola gitmeden algorithm' i analysis'e dönüştürmek.

(b) Biçimlendirme mesafesi *d(x,y)* olan herhangi iki *x* ve *y* dizgisi için, *x*'i *y*' ye, *d(x,y)* maliyetiyle dönüştüren ve hiç 'sol' işlemi olmayan bir *S* dönüştürme işlemleri dizisi olduğunu tartışın.

Çözüm: *x*' i *y*'ye hiç sol işlemi kullanmayan ve maliyeti *d(x,y)* olan işlemlerle dönüştürecek bir *S* dizisi olduğunu çelişki yöntemiyle tartışırız. Farzedin ki böyle bir *S* dizisi olmasın. Bir *S´* dizisini *x*'i *y*' ye *d(x,y)* maliyetiyle ve 'sol' işlemlerini kullanarak dönüştürdüğünü düşünün. *S´* 'deki işlemlerin araya yerleştirdiği harfleri düşünün. Eğer bir harf araya yerleştiriliyor ve sonra siliniyorsa bu durumda her iki işlem de *S´* 'nden çıkarılabilir ve aynı sonucu daha ucuz maliyette verecek bir *S"* üretilebilir. Eğer bir *a* harfi araya yerleştiriliyor ve sonra *b* harfiyle değiştiriliyorsa, bu durumda araya yerleştirme işlemi, *b*'yi araya yerleştirme olarak uygulanabilir ve değiştir işlemi de daha düşük maliyetli bir S" dizisi oluşacak şekilde çıkarılabilir. Eğer bir harf *a* harfiyle değiştiriliyorsa ve sonra da bu *b* harfiyle değiştiriliyorsa bu iki işlem '*b*' yi değiştir' şeklinde uygulanacak tek işleme dönüştürülebilir. Yani her araya yerleştirilen ve değiştirilen karakterler, araya yerleştirme ve değiştirme işlemlerinden sonra hiçbir zaman değiştirilmezler. Bağımlılık yaratan bu işlemleri kaldırdıktan sonra, araya yerleştirme, silme ve değiştirme işlemleri yeniden sıraya sokulabilir ve bu şekilde dönüşümün sonucunu etkilemeden soldan sağa doğru görünürler.

(c) Biçimlendirme mesafesi *d(x,y)*' yi hesaplama probleminin en iyi alt yapıyı kullandığını gösterin. *(İpucu: x* ve *y*' nin tüm son takılarını ele alın. *)*

Çözüm:

x ve *y* dizgilerinin biçimlendirme mesafesini hesaplamanın, alt problemlerin biçimlendirme mesafelerini bulmakla yapılabileceğini gösteririz. Bir maliyet fonksiyonu tanımlayın.

$$c_{xy}(i,j) = d(x, y[1\,..\,i] \| x[j+1\,..\,m]) \qquad \textbf{Denklem (1)}$$

Yani, $c_{xy}\ (i, j)$, *x*'in ilk *j* karakterlerini,*y*' nin ilk *i* karakterlerine dönüştürmenin asgari maliyetidir.

Böylece $d(x, y) = c_{xy}\ (n, m)$. Şimdi *x*' i, *y*' ye, $C(S) = d(x, y)$ maliyetiyle dönüştüren bir $S = (O_1, O_2, \ldots, O_k)$ işlemler dizisini düşünün. S_i, *S'* nin içindeki ilk *i* işlemini içeren *S*' nin altdizisi olsun. Z_i ' de, S_i işlemlerini yaptıktan sonraki ek dizgi olsun; burada $Z_0 = x$ and $Z_k = y$.

Teorem 1 Eğer $C(S_i) = d(x, Z_i)$, *ise;* $C(S_{i-1}) = d(x, z_{i-1})$ olur.

Yani $d(x, Z_i)$' nin en iyi çözümü $d(x, z_{i-1})$ alt problemlerinin en iyi çözümlerini içerir. Bu iddiayı kes-yapıştır kullanarak çelişki yöntemiyle kanıtlayacağız. $C(S_{i-1}) \neq d(x, z_{i-1})$ olduğunu varsayın. Böylece iki durum olur, $C(S_{i-1}) < d(x, z_{i-1})$ veya $C(S_{i-1}) > d(x, z_{i-1})$. Eğer $C < d(x, z_{i-1})$ ise, $d(x, z_{i-1})$' den daha az maliyetli S_{i-1} işlem kullanarak *x*' i Z_{i-1}'e dönüştürebiliriz ama bu bir çelişkidir. Eğer $C(S_{i-1}) > d(x, z_{i-1})$ ise , *x*'i, Z_{i-1}'e, $d(x, z_{i-1})$ maliyetle dönüştüren, *S*' işlemlerinin dizisini S_{i-1} yerine kullanabiliriz. Böylece $S' U o_i$ işlemler dizisi *x*' i *y*' ye şu maliyetle dönüştürür; $C(S' \cup o_i)$.

$$\begin{aligned} C(S' \cup o_i) &= d(x, z_{i-1}) + C(o_i) \\ &< C(S_{i-1}) + C(o_i) \\ &= C(S_i) \\ &= d(x, z_i) \end{aligned}$$

Bunun anlamı $d(x, z_i)$' nin *x* ile z_i arasındaki biçimlendirme mesafesi olmadığıdır ve bu bir çelişkidir. Bu nedenle teorem 1 doğrudur ve biçimlendirme mesafesi problemi en iyi altyapı özlliği gösterir.

(d) $d(x, y)$ 'nin biçimlendirme mesafesinin değerini *x* ve *y*' nin sontakıları cinsinden özyinelemeli olarak tanımlayın. Biçimlendirme mesafesinin nasıl örtüşme altproblemleri oluşturduğunu gösterin.

Çözüm:

Denklem 1' deki $c_{xy}(i,j)$' nin tanımını kullanarak *d(x, y)*'nin biçimlendirme mesafesini hesaplayabiliriz. *d(x, y)*= $c_{xy}(m,n)$ olduğunu hatırlayın. Şık (a)' da 'sol' işlemini kullanmaksızın d(x,y)' yi elde etmek için bir işlemler dizisi bulunduğunu gösterdiğimizden, sadece 4 işlemle, yani 'sağ', 'değiştir(replace)', 'sil' ve 'araya yerleştir(insert) ile ilgilenebiliriz. $c_{xy}(m,n)$' yi özyinelemeli olarak hesaplayabiliriz. *Taban durumu hiç dönüştürme işleminin yapılmamış olduğu durumdur, yani* $c_{xy}(0, 0)=0$.

$$c_{xy}(i,j) = \min \begin{cases} 0 & \text{if } i = 0 \text{ and } j = 0 \\ c_{xy}(i-1,j-1) & \text{if } i > 0 \text{ and } j > 0 \text{ and } x[i-1] = y[j-1] \\ c_{xy}(i-1,j-1)+4 & \text{if } i > 0 \text{ and } j > 0 \\ c_{xy}(i-1,j)+2 & \text{if } i > 0 \\ c_{xy}(i,j-1)+3 & \text{if } j > 0 \end{cases}$$

Denklem (2)

Bu özyinelemeli çözüm örtüşme altproblemleri özelliği gösterir. Örneğin $c_{xy}(i, j)$, $c_{xy}(i - 1, j)$, ve $c_{xy}(i,j - 1)$ hesaplamalarının hepsi $c_{xy}(i - 1, j - 1)$ altprobleminin özyinelemeli olarak hesaplanmasını gerektirir.

(e) *x[1.. m]* 'den *y[1.. n]*'ye biçimlendirme mesafesini hesaplayan bir dinamik programlama algoritmasını açıklayın. (Memolandırılmış özyinelemeli bir algoritma kullanmayın. Algoritmanız klasik, aşağıdan yukarıya, tablo yapısını destekleyen bir algoritma olsun.) Algoritmanızın koşma süresini ve alan gereksinimini çözümleyin.

Çözüm: Her girdisinin $T[i, j] = c_{xy}(i, j)$ olduğu bir T tablosu yapın. $i' \leq i$ ve $j' \leq j$ ise, $c_{xy}(i, j)$'nin her değeri sadece $c_{xy}(i', j')$ 'ye bağımlı olacağından, **Denklem 2**'yi kullanarak *T*'nin girdilerini sıra sıra hesaplayabiliriz:

EDIT-DISTANCE (x[1..m], y[1..n])

```
1  T[0,0] ← 0
2  for i ← 1 to n
3      for j ← 1 to m
4          do T[i,j] ← min { T[i-1,j-1]      if i > 0 and j > 0 and x[i-1] = y[j-1]
                           { T[i-1,j-1] + 4  if i > 0 and j > 0
                           { T[i-1,j] + 2    if i > 0
                           { T[i,j-1] + 3    if j > 0
5  return T[n,m]
```

Bu algoritmanın koşma süresi $\Theta(mn)$. Bu algoritma için $\Theta(mn)$ alan gerekir.

(f) Algoritmanızı dilediğiniz dilde bir bilgiişlem programi olarak uygulayın.[2]
Programınız dinamik programlama kullanarak x ve y gibi iki dizgi arasındaki *d(x, y)* biçimlendirme mesafesini hesaplamalı ve ilgili dönüşüm işlemleri dizisine Tablo 1 'deki stilde baskıya göndermelidir.

[2] Çözümler Java ve Python ile sağlanacaktır

Programınızı aşağıdaki dizgiler üzerinde çalıştırın:

x = "electrical engineering", (elektrik mühendisliği)

y = "computer science". (bilgiişlem bilimi)

Programınızın kaynak kodunu sınıfın web sitesine gönderin ve kaynak kodunuzla sonuçlarınızın baskı kopyasını verin.

Çözüm: Java ve Python ile yapılan çözümlerin kaynak kodu sınıfın web sayfasında bulunabilir. Yukarıdaki girişler dayalı program çıkışı şöyle olur:

```
x: electrical engineering
y: computer science
Edit Distance: 54

Oper    | c |Total| z
initial | 0 |    0 | *electrical engineering
delete  | 2 |    2 | *lectrical engineering
delete  | 2 |    4 | *ectrical engineering
delete  | 2 |    6 | *ctrical engineering
right   | 0 |    6 | c*trical engineering
insert  | 3 |    9 | co*trical engineering
insert  | 3 |   12 | com*trical engineering
insert  | 3 |   15 | comp*trical engineering
insert  | 3 |   18 | compu*trical engineering
right   | 0 |   18 | comput*rical engineering
insert  | 3 |   21 | compute*rical engineering
right   | 0 |   21 | computer*ical engineering
delete  | 2 |   23 | computer*cal engineering
delete  | 2 |   25 | computer*al engineering
delete  | 2 |   27 | computer*l engineering
delete  | 2 |   29 | computer* engineering
right   | 0 |   29 | computer *engineering
delete  | 2 |   31 | computer *ngineering
replace | 4 |   35 | computer s*gineering
replace | 4 |   39 | computer sc*ineering
right   | 0 |   39 | computer sci*neering
delete  | 2 |   41 | computer sci*eering
delete  | 2 |   43 | computer sci*ering
right   | 0 |   43 | computer scie*ring
delete  | 2 |   45 | computer scie*ing
delete  | 2 |   47 | computer scie*ng
right   | 0 |   47 | computer scien*g
insert  | 3 |   50 | computer scienc*g
replace | 4 |   54 | computer science*
```

Bu çözümün tek çözüm olmadığına dikkat edin. Aynı maliyeti olan başka dönüşüm dizileri de olabilir.

Programınızdaki hataları ayıklamanıza yardımcı olmak için sınıf web sitesinde örnek Girdi ve Çıktı Tekstleri verilmiştir. Bu çözümler tek değildir: Aynı maliyeti olan başka dönüşüm dizileri de olabilir. Her zamanki gibi,bu problemin çözümünde işbirliği yapabilirsiniz ama, programı kendiniz yazmalısınız.

(g) Sınıf web sitesinde verilen 3 girdi dosyasıyla programınızı çalıştırın. Her girdi dosyası aşağıdaki 4 satırı içermektedir:

1. Dizgi *x*' deki *m* karakterlerinin sayısı.
2. Dizgi *x.*
3. Dizgi *y'* deki *n* karakterlerinin sayısı.
4. Dizgi *y.*

Her girdi için biçimlendirme mesafesi *d(x,y)*' yi hesaplayın. Problemin bu bölümü için dönüşüm işlemlerinin basılmış halini teslim etmeyin. (web'de aramadan tekstlerin kaynağını belirleyebilirsiniz).

Çözüm:

Input File	*d(x, y)*
Input 1	1816
Input 2	1824
Input 3	1829

(h) Eğer z, bir dizilim kullanılarak uygulanırsa, o zaman ''araya yerleştirme'' ve ''silme'' işlemleri için Θ(n) süresi gereklidir. 5 dönüşüm işleminin her birine, *O(1)* sürede gerçekleştirecek uygun bir veri yapısı tasarlayın.

Çözüm: *L* ve *R* gibi iki yığıt kullanarak tüm dönüşüm işlemleri *O(1)* sürede yapılabilir. Başlangıçta *L* boştur ve *R* tüm *x* karakterlerini sıralı halde içerir.

İşlem	*Uygulama*
left	If not EMPTY(L), then PUSH(R, POP(L))
right	If not EMPTY(R), then PUSH(L, POP(R))
replace by c	If not EMPTY (R), then POP (R), PUSH(L, C)
delete	If not EMPTY *(R),* then POP (R)
insert c	PUSH(L, C)

Her yığıt işlemi O(1) süre gerektirir ve her dönüşüm işlemi sadece O(1) yığıt işlemi gerektirir. Bu nedenle her işlem için O(1) süresi gerekir.

Problem 7-2. GreedSox

GreedSox, popüler bir baseball takımıdır ve sadece bir şeyle ilgilenir: para kazanma. Onlar sizi toplu bilet satışlarını arttırma konusunda danışman olarak kiralamışlar. Şu problemi farketmişler. Bir grup, bir maçı seyretmek istediğinde, grubun tüm üyeleri açık tribünde koltuk istiyor, yoksa gidiyorlar. Gruplar parçalanarak oturtulmadığından dolayı, açık tribün genellikle dolu olmuyor. Oturacak yerler oluyor fakat bütün grup için yeterli olmuyor. Bu durumda da grup oturtulamadığından, GreedSox zarar ediyor.

GreedSox; yeni bir oturma planı için sizin tavsiyenizi istiyor. İlk gelen ilk oturur politikası yerine, GreedSox önce büyük gruplara, sonra küçük gruplara ve en son da kişilere (yani 1' li gruplara) yer ayırmaya karar veriyor.

Size $G[1..m] = [g1, g_2, \ldots, g_m]$, gibi grup kümeleri veriliyor ve burada g_i grubun büyüklüğünü gösteren sayı. Açık tribünde *n* sayıda insanın oturabileceğini varsayın. ADMIT(i) 'nin grup i ' yi kabul ettiği ve REJECT(i)' nin de grup(i)' yi reddettiği aşağıdaki hırslı oturma algoritmasını düşünün.

```
SEAT(G[1..m],n)
1  admitted <— 0          (yerleştirilen)
2  remaining <— n         (kalan)
3  G <- SORT(G)           ▷ Grupları büyükten küçüğe sıralayın.
4  for i <— 1 to m
5      do if G[i] < remaining
6          then ADMIT (i)   (öyleyse yerleştir)
7               remaining <- - remaining — G[i]
8               admitted <—  admitted + G[i]
9          else REJECT (i)  (başkaysa reddet)
10 return admitted      (yerleştirildi döndür)
```

SEAT algoritması (koltuk algoritması) grupları önce boyutuna göre sıralıyor. Sonra gruplar arasında büyükten küçüğe bir döngüye girerek açık tribüncüleri alacak herhangi bir gruba yer veriyor. Kabul edilen insan sayısını çıkışa veriyor.

(a) GreedSox' ın sahipleri haklı, hırslı yerleştirme algoritması hızlı çalışıyor. Hırslı yerleştirme algoritmasının en az *k/2* kişiyi oturtmaya izin vermesi durumunda *G* ve *n* verildiğinde *k* kişinin tribune alınabileceğini gösterin.

Çözüm: SEAT algoritmasının izin vereceği insan sayısı konusunda bir önkuramı kanıtlayarak başlarız.

Önkuram 2 SEAT algoritması; ya boyutu n' ye küçük-eşit olan tüm grupları kabul eder veya n/2' ye büyük-eşit sayıda kişiyi kabul eder.

Kanıt. SEAT algoritmasının n' ye küçük-eşit tüm grupları kabul **etmediğini** düşünün. Yani boyutu $g_i \leq n$ olan bazı grupları SEAT kabul etmez.

Burada düşünülecek iki durum var. Önce $g_i \geq n/2$. Öyleyse algoritma hırslı olduğundan g_i' den daha büyük bazı grupları kabul etmiş olduğunu biliriz; aksi halde g_i ' ye de izin verirdi. Bu nedenle algoritmanın istendiği gibi n/2 kişiden daha fazla kişiyi yerleştirdiği sonucuna varırız.

İkinci durum için $g_i < n/2$ olduğunu varsayın. g_i kabul edilmediğinden biliriz ki bir noktada $kalan < g_i < n/2$ olmuştur. *Kalan*, artan bir değer olmadığından en az n/2 kişinin yerleştirildiğine karar veririz.

Bu önkuramın ilk bakışta SEAT algoritmasının iki-rekabetçi olduğunu savunuruz. Birincisi, eğer SEAT, *n*' ye küçük-eşit bütün gruplara izin veriyorsa bu durumda en iyi yerleştirme algoritmasındaki tam olarak aynı sayıda kişiye izin veriyordur. İkincisi, eğer SEAT en azından *n/2* kişiyi kabul ediyorsa, biliyoruz ki en iyi yerleştirme algoritması en fazla *n* kişiyi oturtabilir. Böylece istendiği gibi $n/2 > k2$.

(b) Maalesef SEAT algoritması mükemmel çalışmaz. SEAT' ın en iyi çözüm olmadığını ters bir örnekle gösterin; asimptotik olarak *n* büyüdükçe, hırslı yerleştirme ile en iyi yerleştirme arasındaki oran *1/2'* ye yaklaşır.

Çözüm: $G = \{(n + 2)/2, n/2, n/2\}$ gruplarını düşünün. Hırslı yerleştirme algoritmasının $(n + 2)/2$ boyutundaki grubu kabul ettiğine ve sonra da başka grupları kabul edemediğine dikkat edin. En iyi algoritma, boyutu *n/2* olan iki grubu da kabul eder, böylece tüm *n* sayıdaki koltuğu doldurur. $((n + 2)/2)/n$, n büyüdükçe asimptotik olarak *1/2* ye yaklaşır.

Sonuçlarınızı GreedSox' un sahiplerine sunduğunuzda aşağıdaki probleme dikkatinizi çekerler:bir bilgisayarın belleğindeki sayıların aksine, gerçek insanları yerlerinden oynatmak zordur. Özellikle kuyrukta bekleyen insanlar ''sıralanmaktan'' hoşlanmazlar. GreedSox' un sahipleri, sizden *G* kümesini değiştirmeyecek bir hırslı yerleştirme algoritması versiyonunu geliştirmenizi isterler. (*G*' nin salt-okunur bellekte depolandığını düşünebilirsiniz). Aşağıdaki algoritmayı öneriyorsunuz.

```
RESEAT (G[1 . . m], n)   (yeniden yerleştir)
 1  admitted  ← 0
 2  remaining  ← n
 3  for j  ←1 to |lg n]
 4       do for i  ← 1 to m
 5            do if G[i] ≥ n/2^j and G[i] ≤   remaining
 6                 then ADMIT (i)
 7                      remaining  ← remaining - G[i]
 8                      admitted  ← admitted + G[i]
 9                 else if G[i] > remaining
10                           then REJECT (i)   (öyleyse reddet)
11  return admitted
```

RESEAT algoritması (yeniden yerleştirme algoritması), grup listelerinden birkaç kez döngüye girer. Birinci döngüde boyutu en az *n/2* olan bir grubu kabul eder. İkinci döngüde boyutu en az *n/4* olan bir grubu kabul eder. Bu şekilde devam ederek, açık tribün dolana kadar, giderek daha küçük gruplara oturma yeri bulur. RESEAT işini bitirdiğinde yerleştirilen kişilerin sayısını çıkışa verir.

(c) G ve n verildiğinde en az *k* sayıda kişinin kabul edildiğini varsayın. RESEAT algoritmasının hala *k/2* kişiyi yerleştireceğini gösterin.

Çözüm:

Bu durumdaki savımız, şık(a)' dakinin çok benzeridir. RESEAT algoritmasının kabul ettiği, kişi sayısı konusundaki aynı önkuramı, yeniden kanıtlamayla işe başlarız.

Önkuram 3 RESEAT algoritması; ya boyutu n' ye küçük-eşit olan tüm grupları kabul eder veya n/2' ye büyük-eşit sayıda kişiyi kabul eder.

Kanıt. RESEAT algoritmasnın n' ye küçük-eşit tüm grupları kabul **etmediğini** düşünün. Yani boyutu $g_i \leq n$ olan bazı grupları RESEAT kabul etmez.

Burada düşünülecek iki durum var. Önce $g_i \geq n/2$ olduğunu varsayalım.. Öyleyse algoritma (j=1 olduğunda) önce n/2' den büyük-eşit olan tüm grupları ele aldığından, n/2' den büyük-eşit bazı grupları kabul ettiğini biliriz; aksi halde j=1 olduğunda g_i'ye de izin verirdi. Bu nedenle algoritmanın istendiği gibi n/2 kişiden daha fazla kişiyi yerleştirdiği sonucuna varırız.

İkinci durum için $g_i < n/2$ olduğunu varsayın. *j= [lg n]* olduğunda, $g \geq n/2^j$ olduğuna dikkat edin . bu nedenle eğer g_i *kabul edilmezse,* $g_i >$ *kalan* olduğu sonucuna varabiliriz. Yani; *kalan* $< g_i <$ *n/2. Kalan,* artan değer olmadığından en az n/2 kişinin oturtulduğu sonucuna varırız.

Önceden olduğu gibi bu önkuram, istendiği biçimde RESEAT algoritmasının iki-rekabetçi olduğunu gösterir.

(d) RESEAT algoritmasının koşma süresi *O(m lg n)* ' dir.Eğer *k* kişi yerleştirilebiliyorsa, O(m) koşma süresinde en az *k/2* kişiyi yerleştirebilen yeni bir algoritma kuramlayın.

Çözüm:

```
FAST-RE SEAT (G[1 . . m], n)   (hızlı yeniden yerleştirme)
 1  admitted <— 0                                  (yerleştirilen)
 2  remaining <— n                                 (kalan)
 3  for i <— 1 to m
 4       do if G[i] ≥ n/2 and G[i] ≤ remaining
 5            then ADMIT (i)
 6                 admitted <— admitted + G[i]
 1                 remaining <— remaining — G[i]
 8  for i <— 1 to m
 9       do if G[i] ≤ remaining
10            then ADMIT (i)
11                 admitted <— admitted + G[i]
12                 remaining <— remaining — G[i]
13  return admitted
```

Bu algoritmanın doğru olduğunu gösteren sav, şık(c)' deki kanıtın temelde aynısıdır. Özellikle yerleştirilen kişi sayısı hakkında aynı önkuramı gösteririz.

Önkuram 4 FAST-RESEAT(hızlı yeniden yerleştirme), algoritması; ya boyutu n' ye küçük-eşit olan tüm grupları kabul eder yada n/2' den büyük-eşit sayıda kişiyi kabul eder.

Kanıt. FAST-RESEAT algoritmasının n' ye küçük-eşit tüm grupları kabul **etmediğini** düşünün. Yani boyutu $g_i \leq n$ olan bazı grupları FAST-RESEAT kabul etmez.

Burada düşünülecek iki durum var. Önce $g_i \geq n/2$ olduğunu varsayalım. Öyleyse algoritma önce n/2' ye büyük-eşit olan tüm grupları ele aldığından, g_i kabul edilmemişse, n/2' ye büyük-eşit bazı grupları kabul ettiğini biliriz; aksi halde g_i ' ye de izin verirdi. Bu nedenle algoritmanın istendiği gibi n/2 kişiden daha fazla kişiyi yerleştirdiği sonucuna varırız.

İkinci durum için $g_i < n/2$ olduğunu varsayın. Bu nedenle eğer g_i *kabul edilmezse,* $g_i >$ *kalan* olduğu sonucuna varabiliriz. Yani; *kalan* $< g_i < n/2$. *Kalan,* artan değer olmadığından en az n/2 kişinin oturtulduğu sonucuna varırız. Önceden olduğu gibi bu önkuram istendiği biçimde, FAST-RESEAT algoritmasının iki-rekabetçi olduğunu gösterir.